

Handling requests with the middleware pipeline

This chapter covers

- What middleware is
- Serving static files using middleware
- Adding functionality using middleware
- Combining middleware to form a pipeline
- Handling exceptions and errors with middleware

In the previous chapter, you had a **whistle-stop tour** of a complete ASP.NET Core application to see how the components come together to create a web application. In this chapter, we'll focus in **on one small subsection**: the middleware pipeline.

In ASP.NET Core, middleware are **C# classes** or **functions** that handle an **HTTP** request or response. They are **chained together**, with the output of one middleware acting as the input to the next middleware, to form a **pipeline**.

The middleware pipeline is one of the most important parts of **configuration** for defining how your application **behaves** and how it **responds** to requests. Understanding **how to build and compose middleware** is **key** to adding functionality to your applications.

In this chapter you'll learn what middleware is and how to use it to create a **pipeline**. You'll see how you can **chain multiple middleware components** together,

with each component adding a **discrete piece of functionality**. The examples in this chapter are limited to using **existing middleware components**, showing how to arrange them in the correct way for your application. In chapter 19 you'll learn how to build your **own middleware components** and **incorporate** them into the pipeline.

We'll begin by looking at the concept of middleware, all the things you can achieve with it, and how a middleware component often maps to a **"cross-cutting concern."** These are the functions of an application that cut across **multiple different layers**. **Logging**, **error handling**, and **security** are **classic** cross-cutting concerns that are all required by many **different parts** of your application. Because all requests pass through the middleware pipeline, it's the preferred location to configure and handle this functionality.

In section 3.2, I'll explain how you can **compose** individual middleware components into a **pipeline**. You'll start out small, with a web app that only displays a **holding page**. From there, you'll learn how to build a **simple static-file server** that returns **requested files** from a folder on disk.

Next, you'll move on to a more complex pipeline containing **multiple** middleware. In this example you'll explore the importance of **ordering** in the middleware pipeline and see how requests are handled when your pipeline contains **more than one middleware**.

In section 3.3 you'll learn how you can use middleware to deal with an important aspect of any application: **error handling**. Errors are a **fact of life** for all applications, so it's important that you account for them when building your app. As well as ensuring that your application doesn't break when an exception is thrown or an error occurs, it's important that users of your application are informed about what went wrong in a **user-friendly way**.

You can handle errors in a few different ways, but they are one of the classic cross-cutting concerns, and **middleware** is well placed to provide the required functionality. In section 3.3 I'll show how you can handle exceptions and errors using middleware provided by **Microsoft**. In particular, you'll learn about **three different components**:

- **DeveloperExceptionPageMiddleware**—Provides quick error feedback when **building** an application
- **ExceptionHandlerMiddleware**—Provides a **user-friendly generic error page** in **production**
- **StatusCodePagesMiddleware**—Converts raw error status codes into **user-friendly error pages**

By combining these pieces of middleware, you can ensure that any errors that do occur in your application **won't leak security details** and won't break your app. On top of that, they will leave users in a **better position** to move on from the error, giving them **as friendly an experience** as possible.

You won't see how to build your **own middleware** in this chapter—instead you'll see that you can go a long way using the **components provided as part of ASP.NET**

Core. Once you understand the **middleware pipeline and its behavior**, it will be much easier to understand **when** and **why** custom middleware is required. With that in mind, let's dive in!

3.1 What is middleware?

The word *middleware* is used in a **variety of contexts** in software development and IT, but it's not a particularly descriptive word—what is middleware?

In ASP.NET Core, middleware are **C# classes¹** that can **handle** an HTTP **request** or **response**. Middleware can

- **Handle** an **incoming HTTP request** by **generating an HTTP response**
- **Process** an incoming HTTP *request*, **modify** it, and pass it on to **another piece of middleware**
- Process an **outgoing HTTP response**, **modify** it, and pass it on to either another piece of middleware or **the ASP.NET Core web server**

You can use middleware in a **multitude** of ways in your own applications. For example, a piece of **logging middleware** might note when a request arrived and then pass it on to another piece of middleware. Meanwhile, an **image-resizing middleware** component might spot an incoming request for an **image** with a **specified size**, generate the requested image, and **send it back to the user** without passing it on.

The most important piece of middleware in most ASP.NET Core applications is the **EndpointMiddleware** class. This class normally **generates** all your **HTML pages** and **API responses** (for Web API applications) and is the **focus** of most of this book. Like the image-resizing middleware, it typically **receives** a request, **generates** a response, and then **sends** it back to the user, as shown in figure 3.1.

DEFINITION This arrangement, where a piece of middleware can call another piece of middleware, which in turn can call another, and so on, is referred to as a **pipeline**. You can think of each piece of middleware as a section of pipe—when you connect all the sections, a request flows through one piece and into the next.

One of the most **common use cases** for middleware is for the **cross-cutting concerns** of your application. These aspects of your application need to **occur for every request**, regardless of the specific path in the request or the resource requested. These include things like

- **Logging** each request
- Adding **standard security headers** to the response
- Associating a request with the **relevant user**
- Setting the **language** for the current request

¹ Technically middleware just needs to be a *function*, as you'll see in chapter 19, but it's very common to implement middleware as a C# class with a single method.

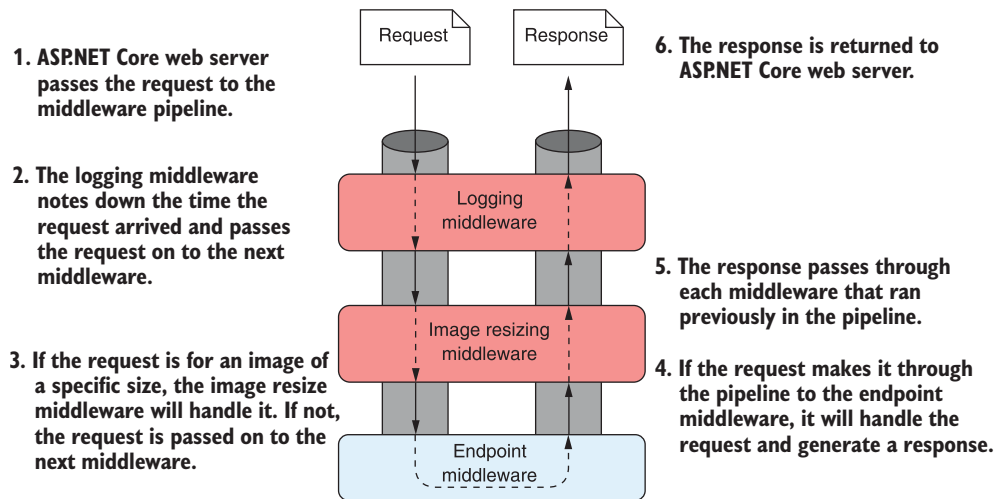


Figure 3.1 Example of a middleware pipeline. Each middleware handles the request and passes it on to the next middleware in the pipeline. After a middleware generates a response, it passes the response back through the pipeline. When it reaches the ASP.NET Core web server, the response is sent to the user's browser.

In each of these examples, the middleware would **receive** a request, **modify** it, and then **pass** the request on to the **next piece** of middleware in the pipeline. Subsequent middleware could use the details added by the earlier middleware to handle the request in some way. For example, in figure 3.2, the **authentication middleware** associates the **request** with a **user**. The authorization middleware uses this detail to **verify** whether the user has **permission** to make that specific request to the application or not.

If the user has permission, the authorization middleware will pass the request on to the endpoint middleware to allow it to generate a response. If the user doesn't have permission, the authorization middleware can short-circuit the pipeline, generating a response directly. It returns the response to the previous middleware before the endpoint middleware has even seen the request.

A key point to glean from this is that the pipeline is *bidirectional*. The request passes through the pipeline in one direction until a piece of middleware generates a response, at which point the response passes *back* through the pipeline, passing through each piece of middleware for a *second* time, until it gets back to the first piece of middleware. Finally, this first/last piece of middleware will pass the response back to the ASP.NET Core web server.

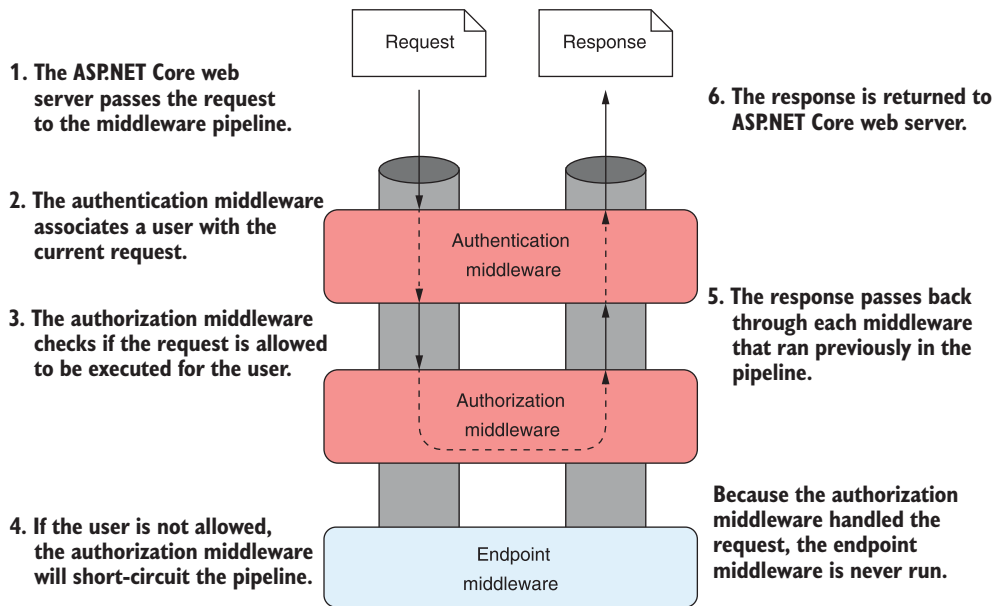


Figure 3.2 Example of a middleware component modifying a request for use later in the pipeline. Middleware can also short-circuit the pipeline, returning a response before the request reaches later middleware.

The **HttpContext** object

I mentioned the `HttpContext` in chapter 2, and it's sitting behind the scenes here too. The ASP.NET Core web server constructs an `HttpContext` for each request, which the ASP.NET Core application uses as a sort of storage box for a single request. Anything that's specific to this particular request and the subsequent response can be associated with and stored in it. This could include properties of the request, request-specific services, data that's been loaded, or errors that have occurred. The web server fills the initial `HttpContext` with details of the original HTTP request and other configuration details and passes it on to the rest of the application.

All middleware has access to the `HttpContext` for a request. It can use this, for example, to determine whether the request contains any user credentials, to identify which page the request is attempting to access, and to fetch any posted data. It can then use these details to determine how to handle the request.

Once the application has finished processing the request, it will update the `HttpContext` with an appropriate response and return it through the middleware pipeline to the web server. The ASP.NET Core web server then converts the representation into a raw HTTP response and sends it back to the reverse proxy, which forwards it to the user's browser.

As you saw in chapter 2, you define the middleware pipeline in code as part of your initial application configuration in `Startup`. You can tailor the middleware pipeline **specifically** to your needs—simple apps may need only a **short pipeline**, whereas large apps with a variety of features may use **much more middleware**. Middleware is the fundamental source of behavior in your application. Ultimately, the middleware pipeline is responsible for **responding** to any HTTP **requests** it receives.

Requests are passed to the middleware pipeline as `HttpContext` objects. As you saw in chapter 2, the ASP.NET Core web server builds an `HttpContext` object from an incoming request, which passes up and down the middleware pipeline. When you're using existing middleware to build a pipeline, this is a detail you'll rarely have to deal with. But, as you'll see in the final section of this chapter, its presence behind the scenes provides a route to exerting extra control over your middleware pipeline.

You can also think of your middleware pipeline as being a series of concentric components, similar to a traditional matryoshka (Russian) doll, as shown in figure 3.3. A request progresses “through” the pipeline by heading deeper into the stack of middleware until a response is returned. The response then returns through the middleware, passing through them in the reverse order to the request.

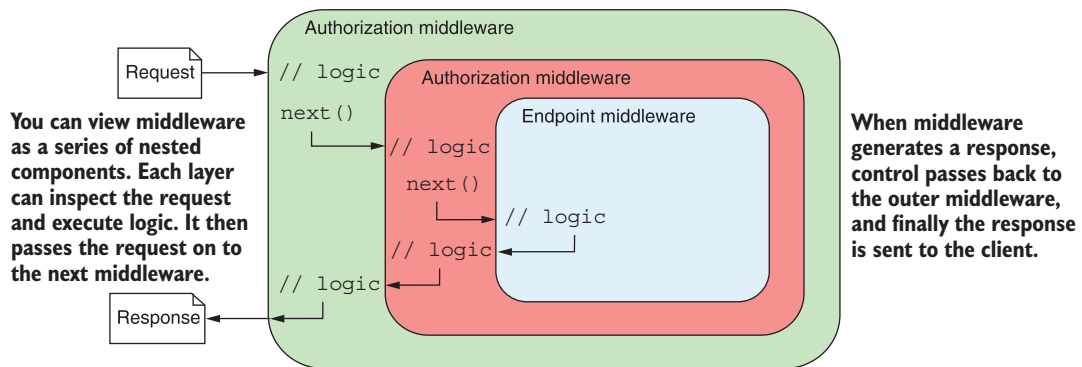


Figure 3.3 You can also think of middleware as being a series of nested components, where a request is sent deeper into the middleware, and the response resurfaces out of it. Each middleware can execute logic before passing the response on to the next middleware and can execute logic after the response has been created, on the way back out of the stack.

Middleware vs. HTTP modules and HTTP handlers

In the previous version of ASP.NET, the concept of a middleware pipeline isn't used. Instead, you have HTTP modules and HTTP handlers.

An *HTTP handler* is a process that runs in response to a request and generates the response. For example, the ASP.NET page handler runs in response to requests for

(continued)

.aspx pages. Alternatively, you could write a custom handler that returns resized images when an image is requested.

HTTP modules handle the cross-cutting concerns of applications, such as security, logging, or session management. They run in response to the lifecycle events that a request progresses through when it's received by the server. Examples of events include `BeginRequest`, `AcquireRequestState`, and `PostAcquireRequestState`.

This approach works, but it's sometimes tricky to reason about which modules will run at which points. Implementing a module requires a relatively detailed understanding of the state of the request at each individual lifecycle event.

The middleware pipeline makes understanding your application far simpler. The pipeline is completely defined in code, specifying which components should run and in which order. Behind the scenes, the middleware pipeline in ASP.NET Core is simply a chain of method calls, where each middleware function calls the next in the pipeline.

That's pretty much all there is to the concept of middleware. In the next section, I'll discuss ways you can combine middleware components to create an application, and how to use middleware to separate the concerns of your application.

3.2 Combining middleware in a pipeline

Generally speaking, each middleware component has a single primary concern. It will handle one aspect of a request only. Logging middleware will only deal with logging the request, authentication middleware is only concerned with identifying the current user, and static-file middleware is only concerned with returning static files.

Each of these concerns is highly focused, which makes the components themselves small and easy to reason about. It also gives your app added flexibility; adding a static-file middleware doesn't mean you're forced into having image-resizing behavior or authentication. Each of these features is an additional piece of middleware.

To build a complete application, you compose multiple middleware components together into a pipeline, as shown in the previous section. Each middleware has access to the original request, plus any changes made by previous middleware in the pipeline. Once a response has been generated, each middleware can inspect and/or modify the response as it passes back through the pipeline before it's sent to the user. This allows you to build complex application behaviors from small, focused components.

In the rest of this section, you'll see how to create a middleware pipeline by composing small components together. Using standard middleware components, you'll learn to create a holding page and to serve static files from a folder on disk. Finally, you'll take another look at the default middleware pipeline you built in chapter 2 and decompose it to understand why it's built like it is.

3.2.1 Simple pipeline scenario 1: **A holding page**

For your first app, and your first middleware pipeline, you'll learn how to create an app consisting of a holding page. This can be useful when you're first setting up your application, to ensure it's processing requests without errors.

TIP Remember, you can view the application code for this book in the GitHub repository at <https://github.com/andrewlock/asp-dot-net-core-in-action-2e>.

In previous chapters, I've mentioned that the ASP.NET Core framework is composed of many small, individual libraries. You typically add a piece of middleware by referencing a package in your application's .csproj project file and configuring the middleware in the `Configure` method of your `Startup` class. Microsoft ships many standard middleware components with ASP.NET Core for you to choose from, and you can also use third-party components from NuGet and GitHub, or you can build your own custom middleware.

NOTE I'll discuss building custom middleware in chapter 19.

Unfortunately, there isn't a definitive list of middleware available, but you can view the source code for all the middleware that comes as part of ASP.NET Core in the main ASP.NET Core GitHub repository (<https://github.com/aspnet/aspnetcore>). You can find most of the middleware in the `src/Middleware` folder, though some middleware is in other folders where it forms part of a larger feature. For example, the authentication and authorization middleware can be found in the `src/Security` folder instead. Alternatively, with a bit of searching on <https://nuget.org> you can often find middleware with the functionality you need.

In this section, you'll see how to create one of the simplest middleware pipelines, consisting of `WelcomePageMiddleware` only. `WelcomePageMiddleware` is designed to quickly provide a sample page when you're first developing an application, as you can see in figure 3.4. You wouldn't use it in a production app, as you can't customize the output, but it's a single, self-contained middleware component you can use to ensure your application is running correctly.

TIP `WelcomePageMiddleware` is included as part of the base ASP.NET Core framework, so you don't need to add a reference to any additional NuGet packages.

Even though this application is simple, the exact same process you've seen before occurs when the application receives an HTTP request, as shown in figure 3.5.

The request passes to the ASP.NET Core web server, which builds a representation of the request and passes it to the middleware pipeline. As it's the first (only!) middleware in the pipeline, `WelcomePageMiddleware` receives the request and must decide how to handle it. The middleware responds by generating an HTML response, no matter what request it receives. This response passes back to the ASP.NET Core web server, which forwards it on to the user to display in their browser.

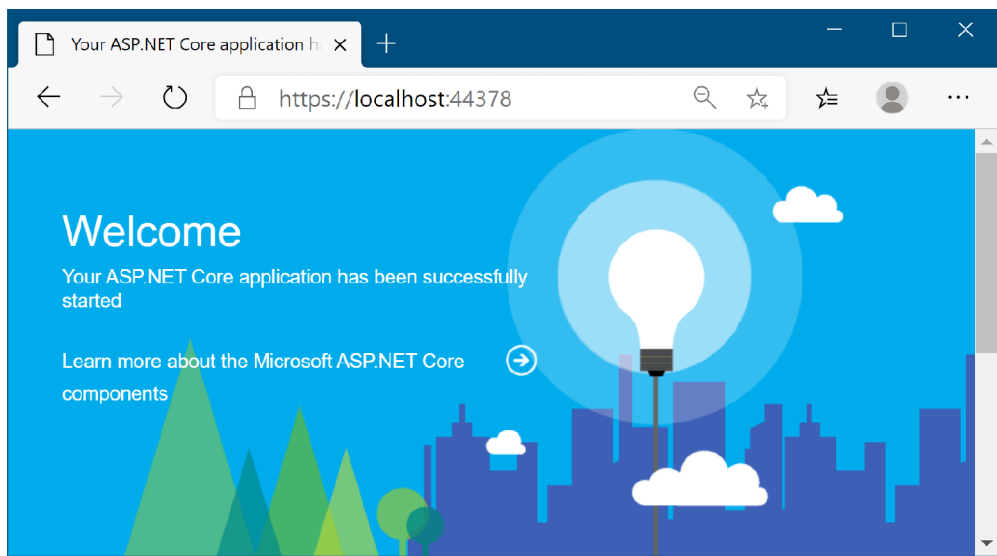


Figure 3.4 The Welcome page middleware response. Every request to the application, at any path, will return the same Welcome page response.

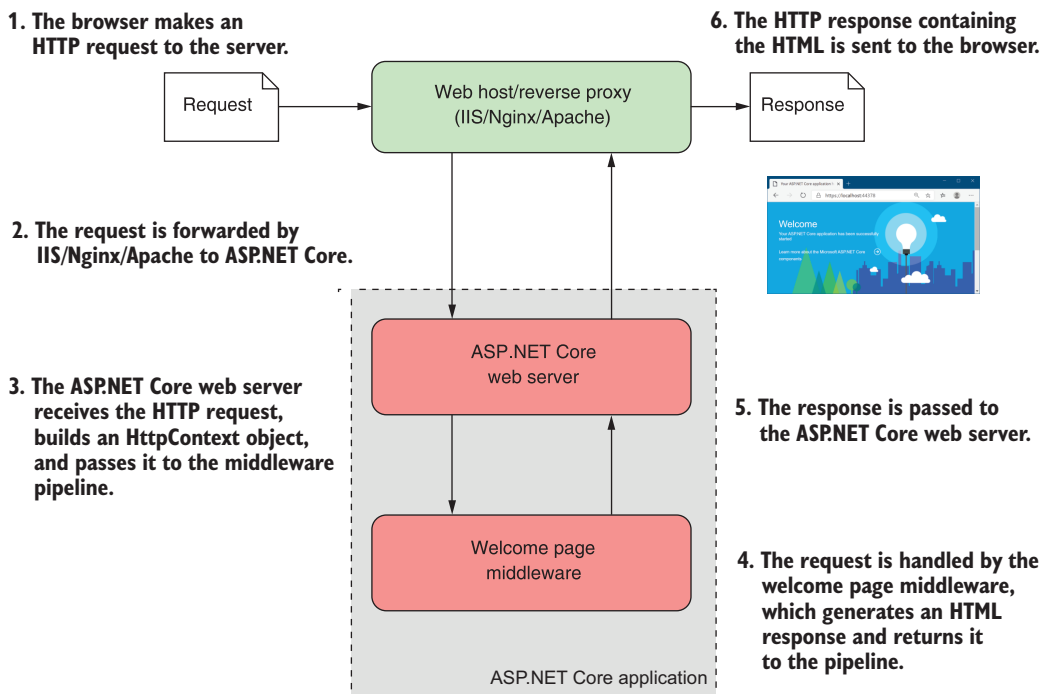


Figure 3.5 `WelcomePageMiddleware` handles a request. The request passes from the reverse proxy to the ASP.NET Core web server and, finally, to the middleware pipeline, which generates an HTML response.

As with all ASP.NET Core applications, you define the middleware pipeline in the `Configure` method of `Startup` by adding middleware to an `IApplicationBuilder` object. To create your first middleware pipeline, consisting of a single middleware component, you need just a single method call.

Listing 3.1 Startup for a Welcome page middleware pipeline

```
using Microsoft.AspNetCore.Builder;
namespace CreatingAHoldingPage
{
    public class Startup
    {
        public void Configure(IApplicationBuilder app)
        {
            app.UseWelcomePage();
        }
    }
}
```

The `Startup` class is very simple for this basic application.

The `Configure` method is used to define the middleware pipeline.

The only middleware in the pipeline

As you can see, the `Startup` class for this application is very simple. The application has no configuration and no services, so `Startup` doesn't have a constructor or a `ConfigureServices` method. The only required method is `Configure`, in which you call `UseWelcomePage`.

You build the middleware pipeline in ASP.NET Core by calling methods on `IApplicationBuilder`, but this interface doesn't define methods like `UseWelcomePage` itself. Instead, these are *extension* methods.

Using extension methods allows you to effectively add functionality to the `IApplicationBuilder` class, while keeping their implementations isolated from it. Under the hood, the methods are typically calling *another* extension method to add the middleware to the pipeline. For example, behind the scenes, the `UseWelcomePage` method adds the `WelcomePageMiddleware` to the pipeline using

```
UseMiddleware<WelcomePageMiddleware>();
```

This convention of creating an extension method for each piece of middleware and starting the method name with `Use` is designed to improve discoverability when adding middleware to your application.² ASP.NET Core includes a lot of middleware as part of the core framework, so you can use IntelliSense in Visual Studio and other IDEs to view all the middleware available, as shown in figure 3.6.

Calling the `UseWelcomePage` method adds the `WelcomePageMiddleware` as the next middleware in the pipeline. Although you're only using a single middleware component here, it's important to remember that the order in which you make calls to

² The downside to this approach is that it can hide exactly which middleware is being added to the pipeline. When the answer isn't clear, I typically search for the source code of the extension method directly on GitHub <https://github.com/aspnet/aspnetcore>.

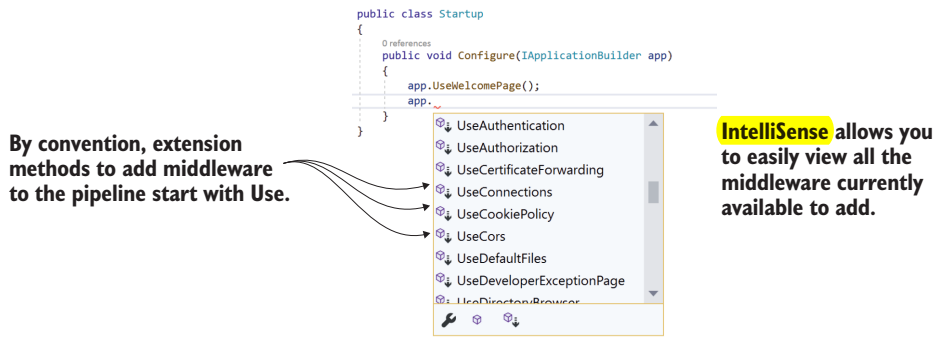


Figure 3.6 IntelliSense makes it easy to view all the middleware available to add to your middleware pipeline.

`IApplicationBuilder` in `Configure` defines the order in which the middleware will run in the pipeline.

WARNING Always take care when adding middleware to the pipeline and consider the order in which it will run. A component can only access data created by middleware that comes before it in the pipeline.

This is the most basic of applications, returning the same response no matter which URL you navigate to, but it shows how easy it is to define your application behavior using middleware. Now we'll make things a little more interesting and return different responses when you make requests to different paths.

3.2.2 *Simple pipeline scenario 2: Handling static files*

In this section, I'll show you how to create one of the simplest middleware pipelines you can use for a full application: a static-file application.

Most web applications, including those with dynamic content, serve a number of pages using static files. Images, JavaScript, and CSS stylesheets are normally saved to disk during development and are served up when requested, normally as part of a full HTML page request.

For now, you'll use `StaticFileMiddleware` to create an application that only serves static files from the `wwwroot` folder when requested, as shown in figure 3.7. In this example, an image called `moon.jpg` exists in the `wwwroot` folder. When you request the file using the `/moon.jpg` path, it's loaded and returned as the response to the request.

If the user requests a file that doesn't exist in the `wwwroot` folder, such as `missing.jpg`, the static-file middleware won't serve a file. Instead, a 404 HTTP error code response will be sent to the user's browser, which will show its default "File Not Found" page, as shown in figure 3.8.

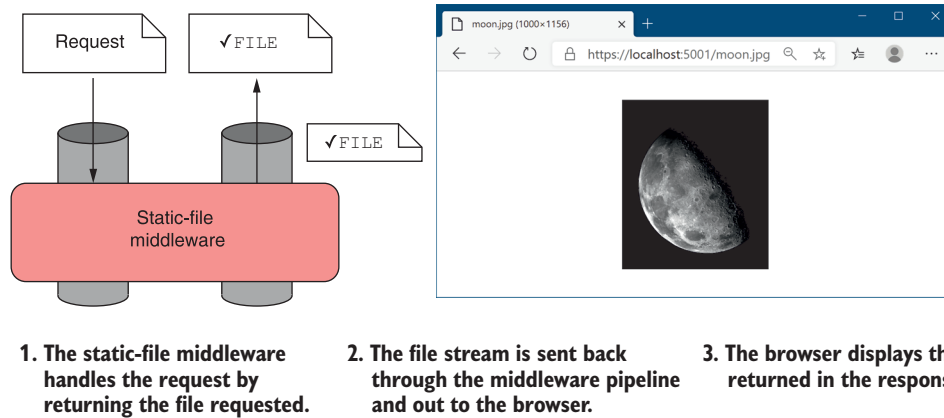


Figure 3.7 Serving a static image file using the static-file middleware

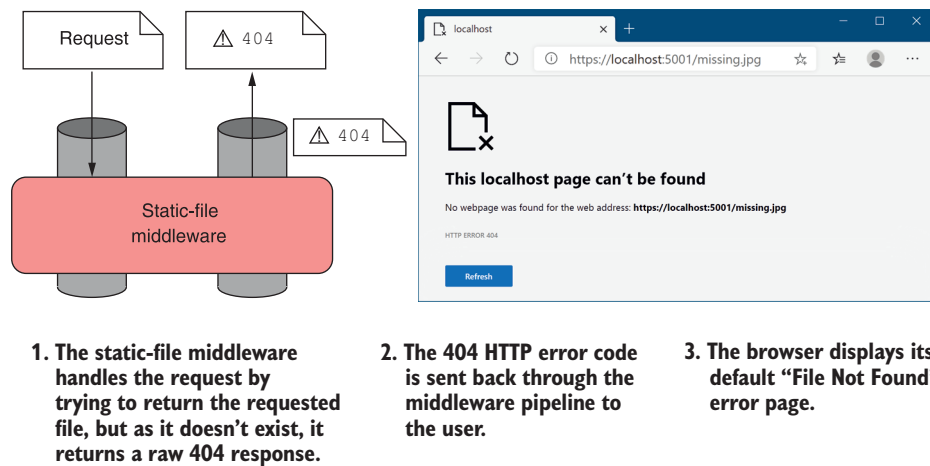


Figure 3.8 Returning a 404 to the browser when a file doesn't exist. The requested file did not exist in the `wwwroot` folder, so the ASP.NET Core application returned a 404 response. The browser, Microsoft Edge in this case, will then show the user a default "File Not Found" error.

NOTE How this page looks will depend on your browser. In some browsers, such as Internet Explorer (IE), you might see a completely blank page.

Building the middleware pipeline for this application is easy. It consists of a single piece of middleware, `StaticFileMiddleware`, as you can see in the following listing. You don't need any services, so configuring the middleware pipeline in `Configure` with `UseStaticFiles` is all that's required.

Listing 3.2 Startup for a static-file middleware pipeline

```
using Microsoft.AspNetCore.Builder;

namespace CreatingAStaticFileWebsite
{
    public class Startup
    {
        public void Configure(IApplicationBuilder app)
        {
            app.UseStaticFiles();
        }
    }
}
```

The Startup class is very simple for this basic static-file application.

The Configure method is used to define the middleware pipeline.

The only middleware in the pipeline

TIP Remember, you can view the application code for this book in the GitHub repository at <https://github.com/andrewlock/asp-dot-net-core-in-action-2e>.

When the application receives a request, the ASP.NET Core web server handles it and passes it to the middleware pipeline. `StaticFileMiddleware` receives the request and determines whether or not it can handle it. If the requested file exists, the middleware handles the request and returns the file as the response, as shown in figure 3.9.

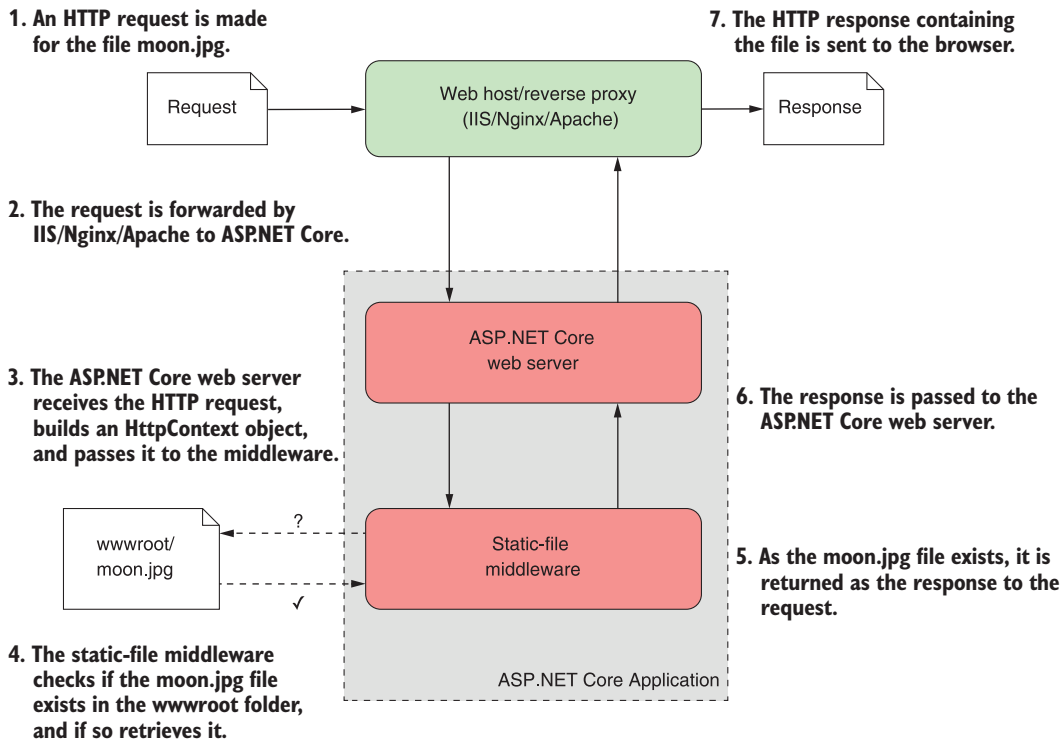


Figure 3.9 `StaticFileMiddleware` handles a request for a file. The middleware checks the `wwwroot` folder to see if the requested `moon.jpg` file exists. The file exists, so the middleware retrieves it and returns it as the response to the web server and, ultimately, to the browser.

If the file doesn't exist, the request effectively passes *through* the static-file middleware unchanged. But wait, you only added one piece of middleware, right? Surely you can't pass the request through to the next middleware if there *isn't* another one?

ASP.NET Core automatically adds a “dummy” piece of middleware to the end of the pipeline. This middleware always returns a 404 response if it's called.

TIP Remember, if no middleware generates a response for a request, the pipeline will automatically return a simple 404 error response to the browser.

HTTP response status codes

Every HTTP response contains a *status code* and, optionally, a *reason phrase* describing the status code. Status codes are fundamental to the HTTP protocol and are a standardized way of indicating common results. A 200 response, for example, means the request was successfully answered, whereas a 404 response indicates that the resource requested couldn't be found.

Status codes are always three digits long and are grouped into five different classes, based on the first digit:

- 1xx—Information. Not often used, provides a general acknowledgment.
- 2xx—Success. The request was successfully handled and processed.
- 3xx—Redirection. The browser must follow the provided link, to allow the user to log in, for example.
- 4xx—Client error. There was a problem with the request. For example, the request sent invalid data or the user isn't authorized to perform the request.
- 5xx—Server error. There was a problem on the server that caused the request to fail.

These status codes typically drive the behavior of a user's browser. For example, the browser will handle a 301 response automatically, by redirecting to the provided new link and making a second request, all without the user's interaction.

Error codes are found in the 4xx and 5xx classes. Common codes include a 404 response when a file couldn't be found, a 400 error when a client sends invalid data (an invalid email address for example), and a 500 error when an error occurs on the server. HTTP responses for error codes may or may not include a response body, which is content to display when the client receives the response.

This basic ASP.NET Core application allows you to easily see the behavior of the ASP.NET Core middleware pipeline and the static-file middleware in particular, but it's unlikely your applications will be as simple as this. It's more likely that static files will form one part of your middleware pipeline. In the next section you'll see how to combine multiple middleware as we look at a simple Razor Pages application.

3.2.3 Simple pipeline scenario 3: A Razor Pages application

By this point, you should have a decent grasp of the middleware pipeline, insofar as understanding that it defines your application's behavior. In this section you'll see how to combine several standard middleware components to form a pipeline. As before, this is performed in the `Configure` method of `Startup` by adding middleware to an `IApplicationBuilder` object.

You'll begin by creating a basic middleware pipeline that you'd find in a typical ASP.NET Core Razor Pages template and then extend it by adding middleware. The output when you navigate to the home page of the application is shown in figure 3.10—identical to the sample application shown in chapter 2.

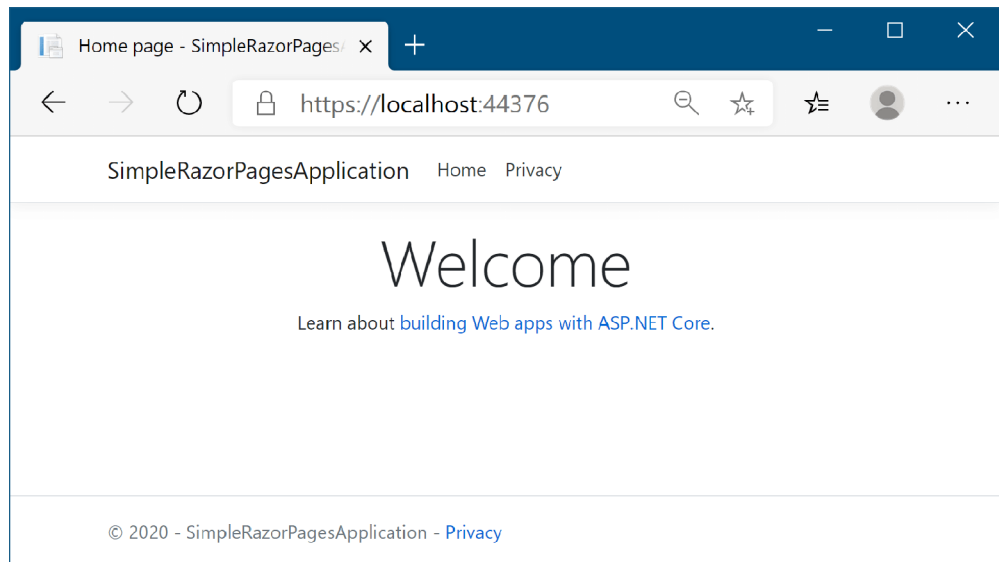


Figure 3.10 A simple Razor Pages application. The application uses only four pieces of middleware: routing middleware to choose a Razor Page to run, endpoint middleware to generate the HTML from a Razor Page, static-file middleware to serve the CSS files, and an exception handler middleware to capture any errors.

Creating this application requires only four pieces of middleware: routing middleware to choose a Razor Page to execute, endpoint middleware to generate the HTML from a Razor Page, static-file middleware to serve the CSS and image files from the `wwwroot` folder, and an exception handler middleware to handle any errors that might occur.

The configuration of the middleware pipeline for the application occurs in the `Configure` method of `Startup`, as always, and is shown in the following listing. As well as the middleware configuration, this listing also shows the call to `AddRazorPages()` in `ConfigureServices`, which is required when using Razor Pages. You'll learn more about service configuration in chapter 10.

Listing 3.3 A basic middleware pipeline for a Razor Pages application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }
    public void Configure(IApplicationBuilder app)
    {
        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

The addition of middleware to `IApplicationBuilder` to form the pipeline should be familiar to you now, but there are a couple of points worth noting in this example. First, all the methods for adding middleware start with `Use`. As I mentioned earlier, this is thanks to the convention of using extension methods to extend the functionality of `IApplicationBuilder`; by prefixing the methods with `Use`, they should be easier to discover.

Another important point about this listing is the order of the `Use` methods in the `Configure` method. The order in which you add the middleware to the `IApplicationBuilder` object is the order in which they're added to the pipeline. This creates a pipeline similar to that shown in figure 3.11.

The exception handler middleware is called first, and it passes the request on to the static-file middleware. The static-file handler will generate a response if the request corresponds to a file; otherwise it will pass the request on to the routing middleware. The routing middleware selects a Razor Page based on the request URL, and the endpoint middleware executes the selected Razor Page. If no Razor Page can handle the requested URL, the automatic dummy middleware returns a 404 response.

NOTE In versions 1.x and 2.x of ASP.NET Core, the routing and endpoint middleware were combined into a single “MVC” middleware. Splitting the responsibilities for routing from execution makes it possible to insert middleware *between* the routing and endpoint middleware. I discuss routing further in chapter 5.

The impact of ordering can most obviously be seen when you have two pieces of middleware that are both listening for the same path. For example, the endpoint middleware in the example pipeline currently responds to a request to the homepage of the application (with the `/` path) by generating the HTML response shown previously in figure 3.10. Figure 3.12 shows what happens if you re-introduce a piece of

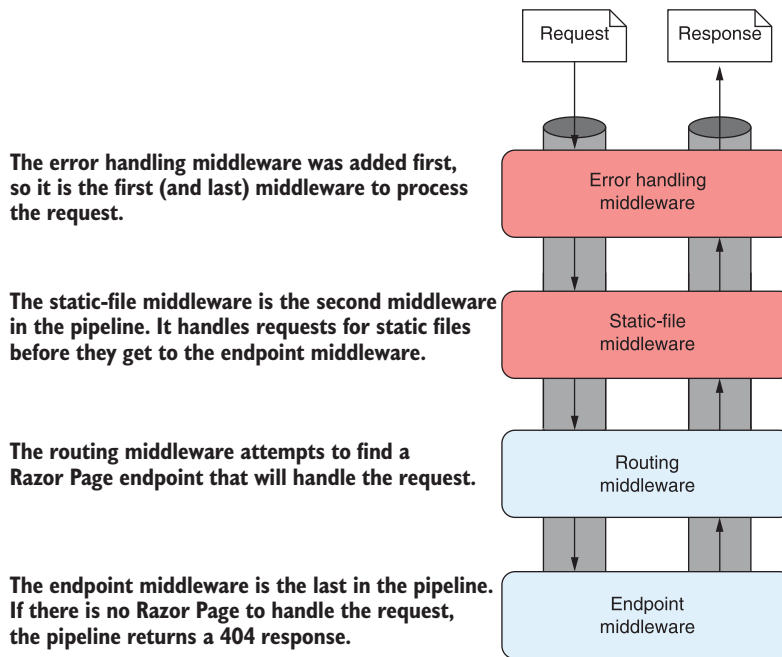


Figure 3.11 The middleware pipeline for the example application in listing 3.3. The order in which you add the middleware to `IApplicationBuilder` defines the order of the middleware in the pipeline.

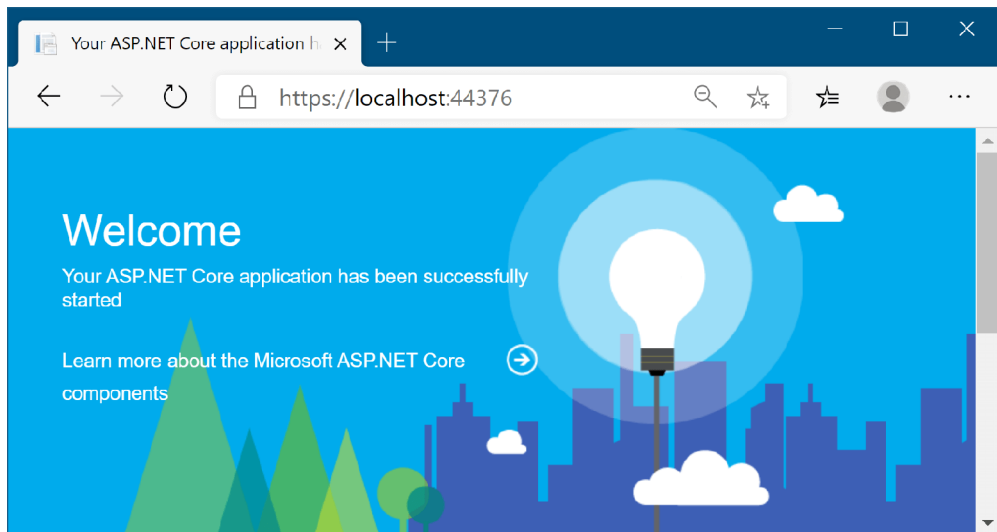


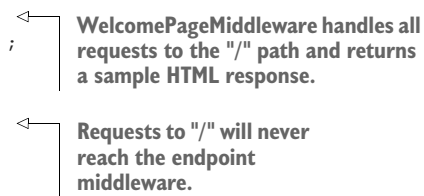
Figure 3.12 The Welcome page middleware response. The Welcome page middleware comes before the endpoint middleware, so a request to the home page returns the Welcome page middleware instead of the Razor Pages response.

middleware you saw previously, `WelcomePageMiddleware`, and configure it to respond to the `"/"` path as well.

As you saw in section 3.2.1, `WelcomePageMiddleware` is designed to return a fixed HTML response, so you wouldn't use it in a production app, but it illustrates the point nicely. In the following listing, it's added to the start of the middleware pipeline and configured to respond only to the `"/"` path.

Listing 3.4 Adding `WelcomePageMiddleware` to the pipeline

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }
    public void Configure(IApplicationBuilder app)
    {
        app.UseWelcomePage("/");
        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```



WelcomePageMiddleware handles all requests to the `"/` path and returns a sample HTML response.

Requests to `"/` will never reach the endpoint middleware.

Even though you know the endpoint middleware can also handle the `"/` path, `WelcomePageMiddleware` is earlier in the pipeline, so it returns a response when it receives the request to `"/`, short-circuiting the pipeline, as shown in figure 3.13. None of the other middleware in the pipeline runs for the request, so none has an opportunity to generate a response.

If you moved `WelcomePageMiddleware` to the end of the pipeline, after the call to `UseEndpoints`, you'd have the opposite situation. Any requests to `"/` would be handled by the endpoint middleware and you'd never see the Welcome page.

TIP You should always consider the order of middleware when adding it to the `Configure` method. Middleware added earlier in the pipeline will run (and potentially return a response) before middleware added later.

All the examples shown so far attempt to handle an incoming request and generate a response, but it's important to remember that the middleware pipeline is bidirectional. Each middleware component gets an opportunity to handle both the incoming request and the outgoing response. The order of middleware is most important for those components that create or modify the outgoing response.

In the previous example, I included `ExceptionHandlerMiddleware` at the start of the application's middleware pipeline, but it didn't seem to do anything. Error handling

1. An HTTP request is made to the URL `http://localhost:49392/`.

6. The HTTP response containing the welcome page is sent to the browser.

2. The Request is forwarded by IIS/Nginx/Apache to ASP.NET Core.

3. The ASP.NET Core web server receives the HTTP request and passes it to the middleware.

4. The Welcome page middleware handles the request. It returns an HTML response, short-circuiting the pipeline.

5. The HTML response is passed back to ASP.NET Core web server.

None of the other middleware is run for the request, so the endpoint middleware does not get a chance to handle the request.

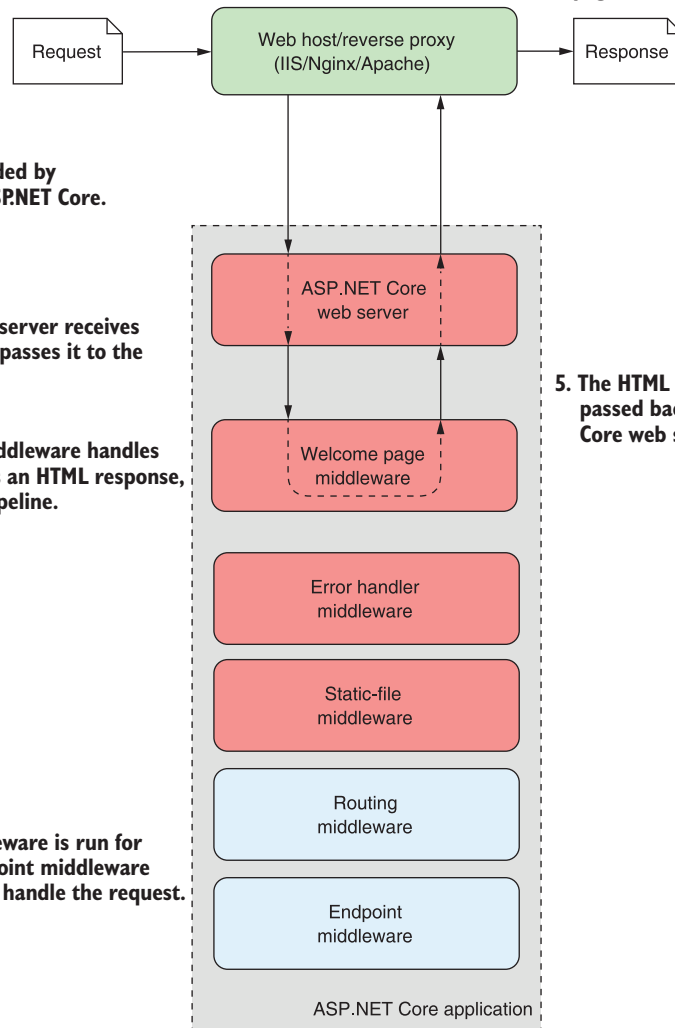


Figure 3.13 Overview of the application handling a request to the `/` path. The welcome page middleware is first in the middleware pipeline, so it receives the request before any other middleware. It generates an HTML response, short-circuiting the pipeline. No other middleware runs for the request.

middleware characteristically ignores the incoming request as it arrives in the pipeline, and instead inspects the outgoing response, only modifying it when an error has occurred. In the next section, I'll detail the types of error handling middleware that are available to use with your application and when to use them.

3.3 Handling errors using middleware

Errors are a fact of life when developing applications. Even if you write perfect code, as soon as you release and deploy your application, users will find a way to break it, whether by accident or intentionally! The important thing is that your application handles these errors gracefully, providing a suitable response to the user, and doesn't cause your whole application to fail.

The design philosophy for ASP.NET Core is that every feature is opt-in. So, as error handling is a feature, you need to explicitly enable it in your application. Many different types of errors could occur in your application, and there are many different ways to handle them, but in this section I focus on two: exceptions and error status codes.

Exceptions typically occur whenever you find an unexpected circumstance. A typical (and highly frustrating) exception you'll no doubt have experienced before is `NullReferenceException`, which is thrown when you attempt to access an object that hasn't been initialized.³ If an exception occurs in a middleware component, it propagates up the pipeline, as shown in figure 3.14. If the pipeline doesn't handle the exception, the web server will return a 500 status code back to the user.

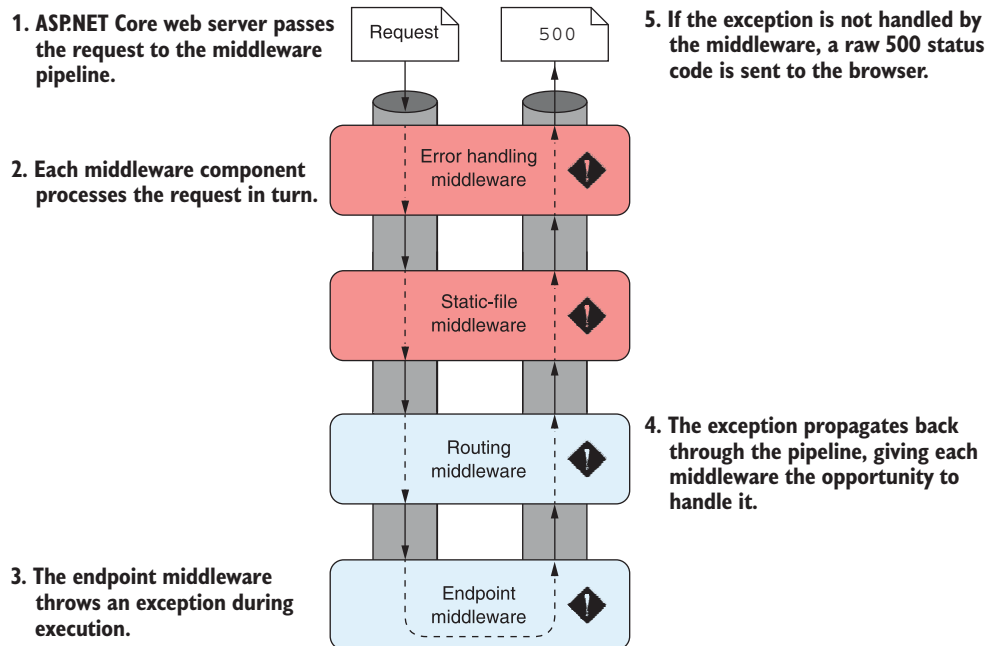


Figure 3.14 An exception in the endpoint middleware propagates through the pipeline. If the exception isn't caught by middleware earlier in the pipeline, then a 500 "Server error" status code is sent to the user's browser.

³ C# 8.0 introduced non-nullable reference types. These provide a way to handle null values more clearly, with the promise of finally ridding .NET of `NullReferenceExceptions`! It will take a while for that to come true, as the feature is opt-in and requires library support, but the future's bright. See the documentation to get started: <http://mng.bz/7V0g>.

In some situations, an error won't cause an exception. Instead, middleware might generate an error status code. One such case is when a requested path isn't handled. In that situation, the pipeline will return a 404 error, which results in a generic, unfriendly page being shown to the user, as you saw in figure 3.8. Although this behavior is “correct,” it doesn't provide a great experience for users of your application.

Error handling middleware attempts to address these problems by modifying the response before the app returns it to the user. Typically, error handling middleware either returns details of the error that occurred, or it returns a generic, but friendly, HTML page to the user. You should always place error handling middleware early in the middleware pipeline to ensure it will catch any errors generated in subsequent middleware, as shown in figure 3.15. Any responses generated by middleware earlier in the pipeline than the error handling middleware can't be intercepted.

The remainder of this section shows several types of error handling middleware that are available for use in your application. They are available as part of the base ASP.NET Core framework, so you don't need to reference any additional NuGet packages to use them.

3.3.1 *Viewing exceptions in development: `DeveloperExceptionPage`*

When you're developing an application, you typically want access to as much information as possible when an error occurs somewhere in your app. For that reason, Microsoft provides `DeveloperExceptionPageMiddleware`, which can be added to your middleware pipeline using

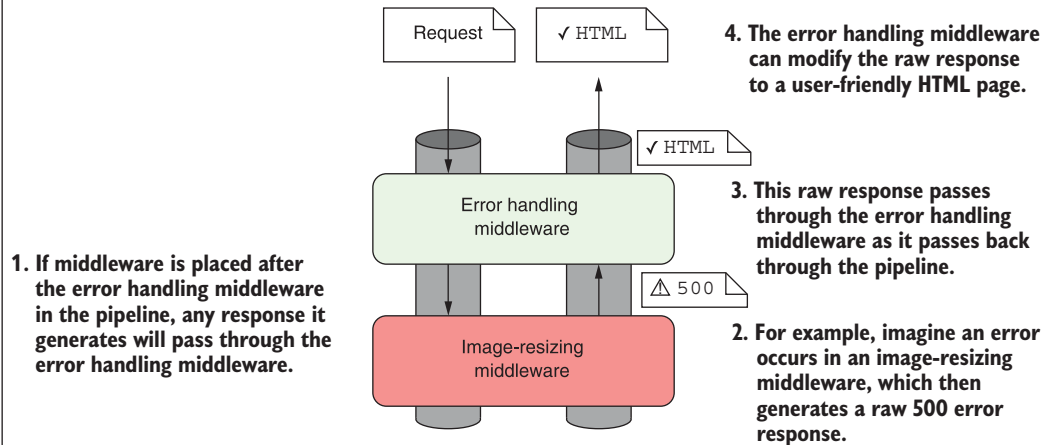
```
app.UseDeveloperExceptionPage();
```

When an exception is thrown and propagates up the pipeline to this middleware, it will be captured. The middleware then generates a friendly HTML page, which it returns with a 500 status code to the user, as shown in figure 3.16. This page contains a variety of details about the request and the exception, including the exception stack trace, the source code at the line the exception occurred, and details of the request, such as any cookies or headers that had been sent.

Having these details available when an error occurs is invaluable for debugging a problem, but they also represent a security risk if used incorrectly. You should never return more details about your application to users than absolutely necessary, so you should only ever use `DeveloperExceptionPage` when developing your application. The clue is in the name!

WARNING Never use the developer exception page when running in production. Doing so is a security risk as it could publicly reveal details about your application's code, making you an easy target for attackers.

Error handling middleware first in the pipeline



Static-file middleware first in the pipeline

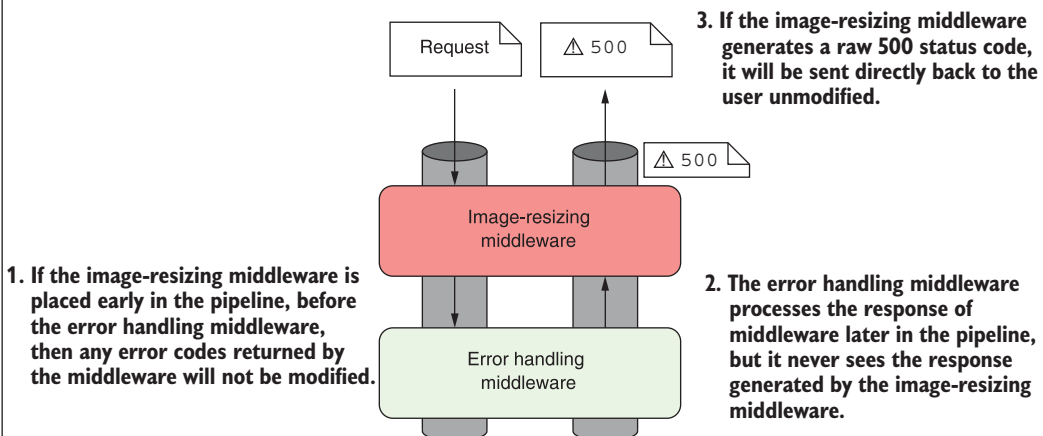


Figure 3.15 Error handling middleware should be placed early in the pipeline to catch raw status code errors. In the first case, the error handling middleware is placed before the image-resizing middleware, so it can replace raw status code errors with a user-friendly error page. In the second case, the error handling middleware is placed after the image-resizing middleware, so raw error status codes can't be modified.

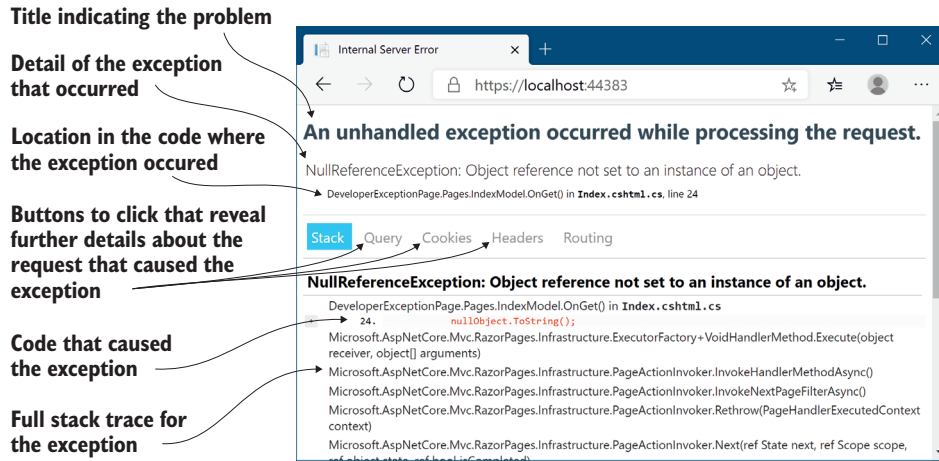


Figure 3.16 The developer exception page shows details about the exception when it occurs during the process of a request. The location in the code that caused the exception, the source code line itself, and the stack trace are all shown by default. You can also click the Query, Cookies, Headers, or Routing buttons to reveal further details about the request that caused the exception.

If the developer exception page isn't appropriate for production use, what should you use instead? Luckily, there's another general-purpose error handling middleware you *can* use in production. It's one that you've already seen and used: `ExceptionHandlerMiddleware`.

3.3.2 Handling exceptions in production: `ExceptionHandlerMiddleware`

The developer exception page is handy when developing your applications, but you shouldn't use it in production as it can leak information about your app to potential attackers. You still want to catch errors, though; otherwise users will see unfriendly error pages or blank pages, depending on the browser they're using.

You can solve this problem by using `ExceptionHandlerMiddleware`. If an error occurs in your application, the user will be presented with a custom error page that's consistent with the rest of the application, but that only provides the necessary details about the error. For example, a custom error page, such as the one shown in figure 3.17, can keep the look and feel of the application by using the same header, displaying the currently logged-in user, and displaying an appropriate message to the user instead of the full details of the exception.

If you were to peek at the `Configure` method of almost any ASP.NET Core application, you'd almost certainly find the developer exception page used in combination with `ExceptionHandlerMiddleware` in a similar manner to that shown in listing 3.5.

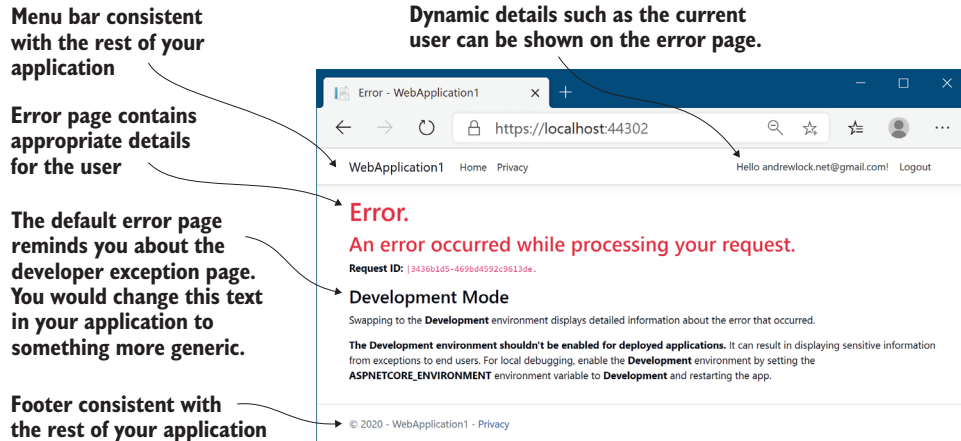


Figure 3.17 A custom error page created by `ExceptionHandlerMiddleware`. The custom error page can keep the same look and feel as the rest of the application by reusing elements such as the header and footer. More importantly, you can easily control the error details displayed to users.

Listing 3.5 Configuring exception handling for development and production

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    // additional middleware configuration
}
```

Annotations for Listing 3.5:

- Configure a different pipeline when running in development. (points to `env.IsDevelopment()`)
- The developer exception page should only be used when running in development mode. (points to `app.UseDeveloperExceptionPage()`)
- When in production, `ExceptionHandlerMiddleware` is added to the pipeline. (points to `app.UseExceptionHandler("/Error")`)

As well as demonstrating how to add `ExceptionHandlerMiddleware` to your middleware pipeline, this listing shows that it's perfectly acceptable to configure different middleware pipelines depending on the environment when the application starts up. You could also vary your pipeline based on other values, such as settings loaded from configuration.

NOTE You'll see how to use configuration values to customize the middleware pipeline in chapter 11.

When adding `ExceptionHandlerMiddleware` to your application, you'll typically provide a path to the custom error page that will be displayed to the user. In the example in listing 3.5, you used an error handling path of `"/Error"`:

```
app.UseExceptionHandler("/Error");
```


`ExceptionHandlerMiddleware` will invoke this path after it captures an exception, in order to generate the final response. The ability to dynamically generate a response is a key feature of `ExceptionHandlerMiddleware`—it allows you to re-execute a middleware pipeline in order to generate the response sent to the user.

Figure 3.18 shows what happens when `ExceptionHandlerMiddleware` handles an exception. It shows the flow of events when the `Index.chstml` Razor Page generates an exception when a request is made to the `"/` path. The final response returns an error status code but also provides an HTML response to display to the user, using the `"/Error"` path.

The sequence of events when an exception occurs somewhere in the middleware pipeline after `ExceptionHandlerMiddleware` is as follows:

- 1 A piece of middleware throws an exception.
- 2 `ExceptionHandlerMiddleware` catches the exception.
- 3 Any partial response that has been defined is cleared.
- 4 The middleware overwrites the request path with the provided error handling path.
- 5 The middleware sends the request back down the pipeline, as though the original request had been for the error handling path.
- 6 The middleware pipeline generates a new response as normal.
- 7 When the response gets back to `ExceptionHandlerMiddleware`, it modifies the status code to a 500 error and continues to pass the response up the pipeline to the web server.

The main advantage that re-executing the pipeline brings is the ability to have your error messages integrated into your normal site layout, as shown previously in figure 3.17. It's certainly possible to return a fixed response when an error occurs, but you wouldn't be able to have a menu bar with dynamically generated links or display the current user's name in the menu. By re-executing the pipeline, you can ensure that all the dynamic areas of your application are correctly integrated, as if the page was a standard page of your site.

NOTE You don't need to do anything other than add `ExceptionHandlerMiddleware` to your application and configure a valid error handling path to enable re-executing the pipeline. The middleware will catch the exception and re-execute the pipeline for you. Subsequent middleware will treat the re-execution as a new request, but previous middleware in the pipeline won't be aware anything unusual happened.

Re-executing the middleware pipeline is a great way to keep consistency in your web application for error pages, but there are some gotchas to be aware of. First, middleware can only modify a response generated further down the pipeline if the response *hasn't yet been sent to the client*. This can be a problem if, for example, an error occurs while ASP.NET Core is sending a static file to a client. In that case, where bytes have

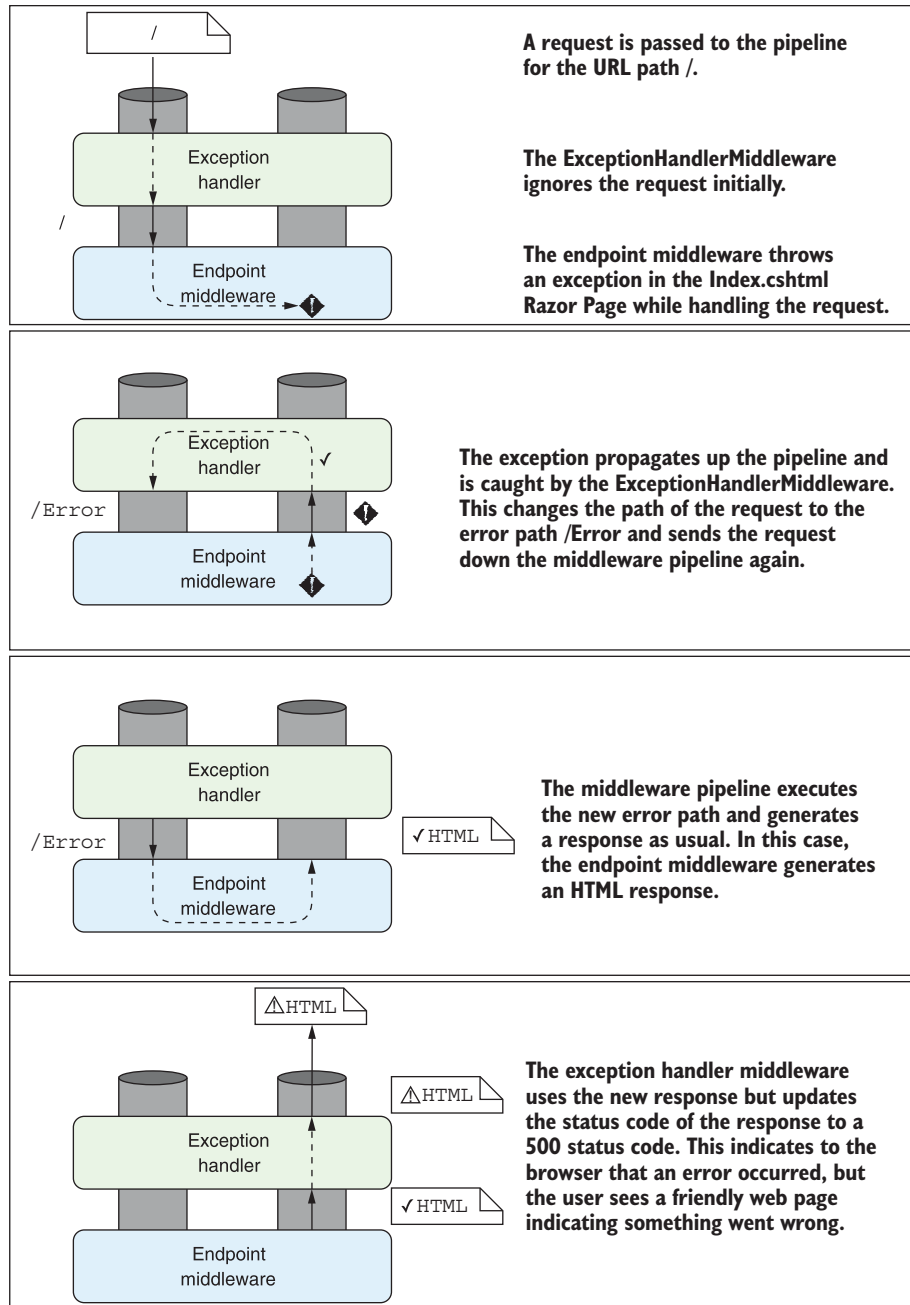


Figure 3.18 `ExceptionHandlerMiddleware` handling an exception to generate an HTML response. A request to the `/` path generates an exception, which is handled by the middleware. The pipeline is re-executed using the `/Error` path to generate the HTML response.

already begun to be sent, the error handling middleware won't be able to run, as it can't reset the response. Generally speaking, there's not a lot you can do about this issue, but it's something to be aware of.

A more common problem occurs when the error handling path throws an error during the re-execution of the pipeline. Imagine there's a bug in the code that generates the menu at the top of the page:

- 1 When the user reaches your homepage, the code for generating the menu bar throws an exception.
- 2 The exception propagates up the middleware pipeline.
- 3 When reached, `ExceptionHandlerMiddleware` captures it and the pipe is re-executed using the error handling path.
- 4 When the error page executes, it attempts to generate the menu bar for your app, which again throws an exception.
- 5 The exception propagates up the middleware pipeline.
- 6 `ExceptionHandlerMiddleware` has already tried to intercept a request, so it will let the error propagate all the way to the top of the middleware pipeline.
- 7 The web server returns a raw 500 error, as though there was no error handling middleware at all.

Thanks to this problem, it's often good practice to make your error handling pages as simple as possible, to reduce the possibility of errors occurring.

WARNING If your error handling path generates an error, the user will see a generic browser error. It's often better to use a static error page that will always work, rather than a dynamic page that risks throwing more errors.

`ExceptionHandlerMiddleware` and `DeveloperExceptionPageMiddleware` are great for catching exceptions in your application, but exceptions aren't the only sort of errors you'll encounter. In some cases, your middleware pipeline will return an HTTP error status code in the response. It's important to handle both exceptions and error status codes to provide a coherent user experience.

3.3.3 *Handling other errors: `StatusCodePagesMiddleware`*

Your application can return a wide range of HTTP status codes that indicate some sort of error state. You've already seen that a 500 "server error" is sent when an exception occurs and isn't handled and that a 404 "file not found" error is sent when a URL isn't handled by any middleware. 404 errors, in particular, are common, often occurring when a user enters an invalid URL.

TIP As well as indicating a completely unhandled URL, 404 errors are often used to indicate that a specific requested object was not found. For example, a request for the details of a product with an ID of 23 might return a 404 if no such product exists.

If you don't handle these status codes, users will see a generic error page, such as in figure 3.19, which may leave many confused and thinking your application is broken. A better approach would be to handle these error codes and return an error page that's in keeping with the rest of your application or, at the very least, doesn't make your application look broken.

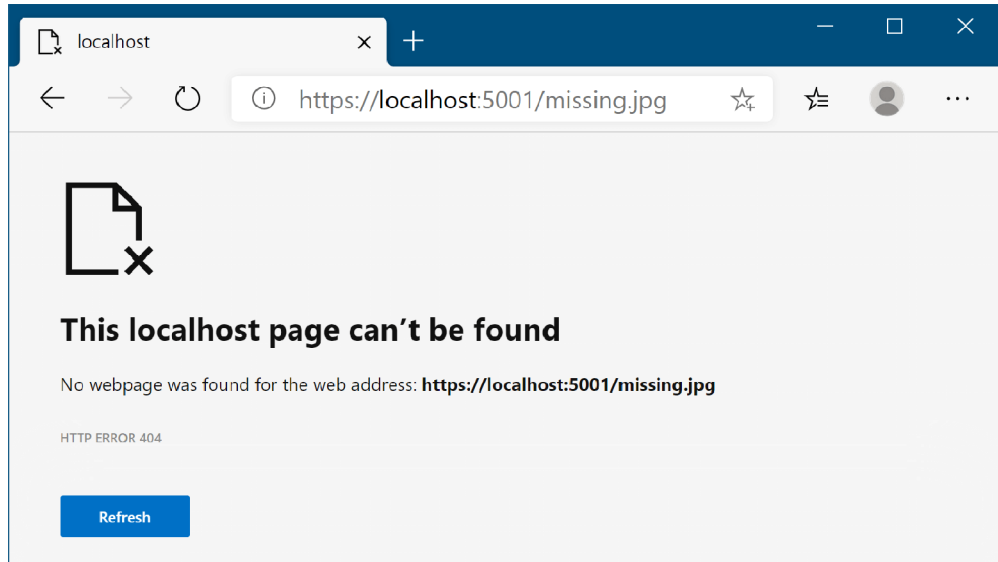


Figure 3.19 A generic browser error page. If the middleware pipeline can't handle a request, it will return a 404 error to the user. The message is of limited usefulness to users and may leave many confused or thinking your web application is broken.

Microsoft provides `StatusCodePagesMiddleware` for handling this use case. As with all error handling middleware, you should add it early in your middleware pipeline, as it will only handle errors generated by later middleware components.

You can use the middleware a number of different ways in your application. The simplest approach is to add the middleware to your pipeline without any additional configuration, using

```
app.UseStatusCodePages();
```

With this method, the middleware will intercept any response that has an HTTP status code that starts with 4xx or 5xx and has no response body. For the simplest case, where you don't provide any additional configuration, the middleware will add a plain text response body, indicating the type and name of the response, as shown in figure 3.20.

This is arguably worse than the default message at this point, but it is a starting point for providing a more consistent experience to users.

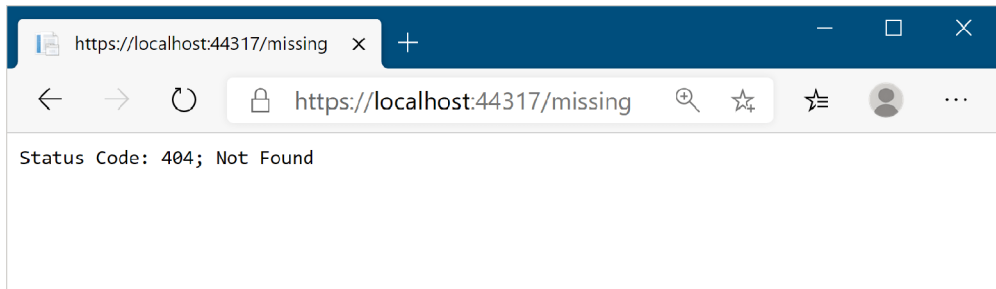


Figure 3.20 Status code error page for a 404 error. You generally won't use this version of the middleware in production as it doesn't provide a great user experience, but it demonstrates that the error codes are being correctly intercepted.

A more typical approach to using `StatusCodePagesMiddleware` in production is to re-execute the pipeline when an error is captured, using a similar technique to the `ExceptionHandlerMiddleware`. This allows you to have dynamic error pages that fit with the rest of your application. To use this technique, replace the call to `UseStatusCodePages` with the following extension method:

```
app.UseStatusCodePagesWithReExecute("/{0}");
```

This extension method configures `StatusCodePagesMiddleware` to re-execute the pipeline whenever a 4xx or 5xx response code is found, using the provided error handling path. This is similar to the way `ExceptionHandlerMiddleware` re-executes the pipeline, as shown in figure 3.21.

Note that the error handling path `"/{0}"` contains a format string token, `{0}`. When the path is re-executed, the middleware will replace this token with the status code number. For example, a 404 error would re-execute the `/404` path. The handler for the path (typically a Razor Page) has access to the status code and can optionally tailor the response, depending on the status code. You can choose any error handling path, as long as your application knows how to handle it.

NOTE You'll learn about how routing maps request paths to Razor Pages in chapter 5.

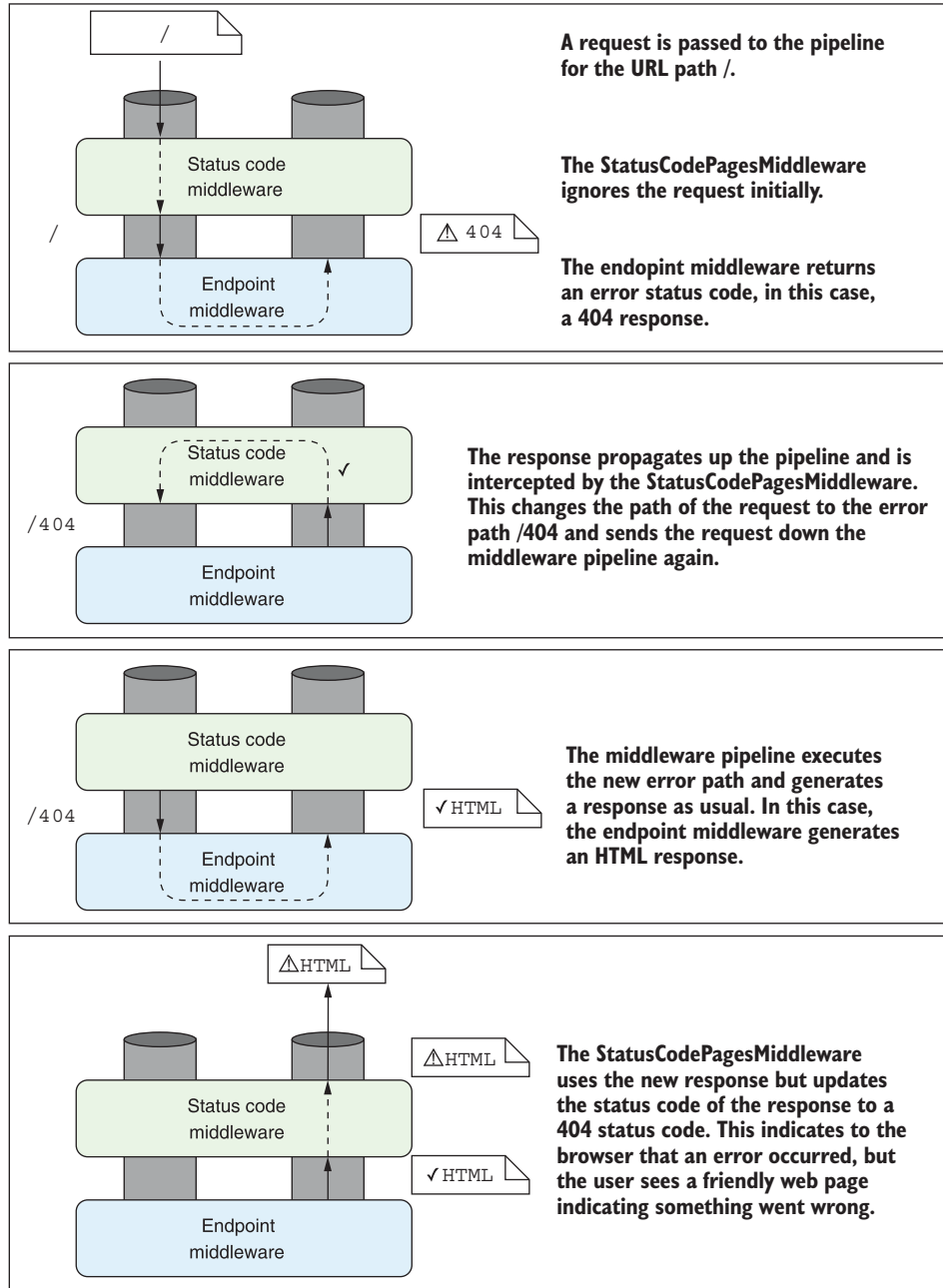


Figure 3.21 `StatusCodePagesMiddleware` re-executing the pipeline to generate an HTML body for a 404 response. A request to the `/` path returns a 404 response, which is handled by the status code middleware. The pipeline is re-executed using the `/404` path to generate the HTML response.

With this approach in place, you can create different error pages for different error codes, such as the 404-specific error page shown in figure 3.22. This technique ensures your error pages are consistent with the rest of your application, including any dynamically generated content, while also allowing you to tailor the message for common errors.

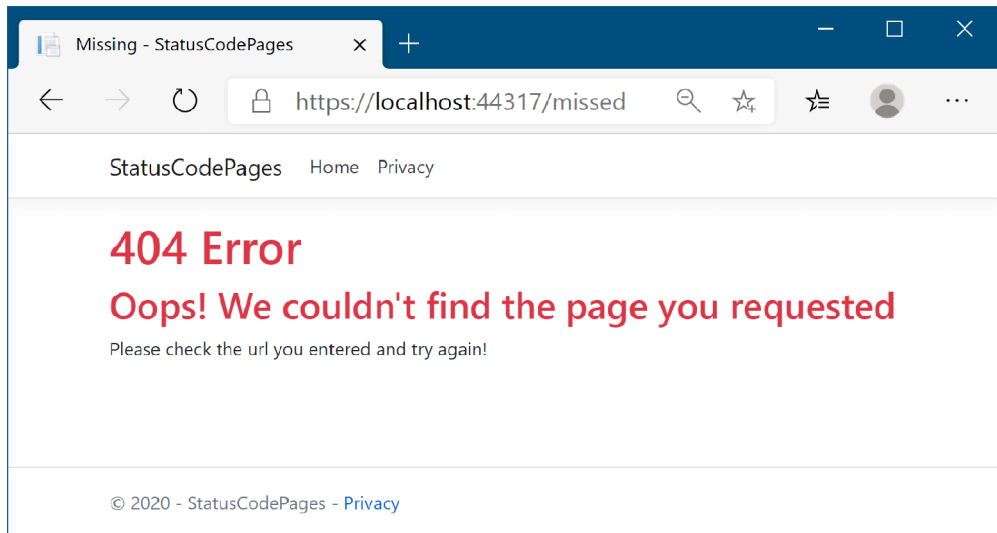


Figure 3.22 An error status code page for a missing file. When an error code is detected (in this case, a 404 error), the middleware pipeline is re-executed to generate the response. This allows dynamic portions of your web page to remain consistent on error pages.

WARNING As before, when re-executing the pipeline, you must be careful your error handling path doesn't generate any errors.

You can use `StatusCodePagesMiddleware` in combination with other exception handling middleware by adding both to the pipeline. `StatusCodePagesMiddleware` will only modify the response if no response body has been written. So if another component, such as `ExceptionHandlerMiddleware`, returns a message body along with an error code, it won't be modified.

NOTE `StatusCodePagesMiddleware` has additional overloads that let you execute custom middleware when an error occurs, instead of re-executing a Razor Pages path.

Error handling is essential when developing any web application; errors happen, and you need to handle them gracefully. But depending on your application, you may not always want your error handling middleware to generate HTML pages.

3.3.4 Error handling middleware and Web APIs

ASP.NET Core isn't only great for creating user-facing web applications, it's also great for creating HTTP services that can be accessed from another server application, from a mobile app, or from a user's browser when running a client-side single-page application. In all these cases, you probably won't be returning HTML to the client, but rather XML or JSON.

In that situation, if an error occurs, you probably don't want to be sending back a big HTML page saying, "Oops, something went wrong." Returning an HTML page to an application that's expecting JSON could easily break it unexpectedly. Instead, the HTTP 500 status code and a JSON body describing the error is more useful to a consuming application. Luckily, ASP.NET Core allows you to do exactly this when you create Web API controllers.

NOTE I discuss MVC and Web API controllers in chapter 4. I discuss Web APIs and handling errors in detail in chapter 9.

That brings us to the end of middleware in ASP.NET Core for now. You've seen how to use and compose middleware to form a pipeline, as well as how to handle errors in your application. This will get you a long way when you start building your first ASP.NET Core applications. Later you'll learn how to build your own custom middleware, as well as how to perform complex operations on the middleware pipeline, such as forking it in response to specific requests.

In the next chapter, you'll look in more depth at Razor Pages, and at how they can be used to build websites. You'll also learn about the MVC design pattern, its relationship with Razor Pages in ASP.NET Core, and when to choose one approach over the other.

Summary

- Middleware has a similar role to HTTP modules and handlers in ASP.NET but is more easily reasoned about.
- Middleware is composed in a pipeline, with the output of one middleware passing to the input of the next.
- The middleware pipeline is two-way: requests pass through each middleware on the way in, and responses pass back through in the reverse order on the way out.
- Middleware can short-circuit the pipeline by handling a request and returning a response, or it can pass the request on to the next middleware in the pipeline.
- Middleware can modify a request by adding data to, or changing, the `HttpContext` object.
- If an earlier middleware short-circuits the pipeline, not all middleware will execute for all requests.
- If a request isn't handled, the middleware pipeline will return a 404 status code.

- The order in which middleware is added to `IApplicationBuilder` defines the order in which middleware will execute in the pipeline.
- The middleware pipeline can be re-executed, as long as a response's headers haven't been sent.
- When it's added to a middleware pipeline, `StaticFileMiddleware` will serve any requested files found in the `wwwroot` folder of your application.
- `DeveloperExceptionPageMiddleware` provides a lot of information about errors when developing an application, but it should never be used in production.
- `ExceptionHandlerMiddleware` lets you provide user-friendly custom error handling messages when an exception occurs in the pipeline. It is safe for use in production, as it does not expose sensitive details about your application.
- `StatusCodePagesMiddleware` lets you provide user-friendly custom error handling messages when the pipeline returns a raw error response status code.
- Microsoft provides some common middleware, and there are many third-party options available on NuGet and GitHub.